UCN, University College of Northern Denmark
IT-Programme
Degree in Computer Science
CSC-CSD-S211

Mini-Project Persistence

Adam Górecki, Andrej Piecka, Aziz Kadem,
Cosmin Raita, Barnabás Selymes

21-03-2022

UCN, University College of Northern Denmark

IT-programme

AP Degree in Computer Science

Class: CSC-CSD-S211

Participants:

Adam Górecki
Andrej Piecka
Aziz Kadem
Cosmin Raita
Barnabás Selymes

Supervisors: Gianna Belle, Karsten Jeppesen

Repository path: https://github.com/PieckaAndrej/Mini_Project_Persistence.git
Repository number: 9fe2cde
Normal pages/characters: 6.69 pages/16079 characters[1]

[1] A normal page is defined as 2.400 characters incl. whitespaces and footnotes. Frontpage, title page, table of contents, literature list and appendices are not included [2]

Table of contents

Abstract/Resume

In this report we are covering the design and implementation of a system based on a given project case. The aim of this project is to help us get a better understanding of how to work with databases, by implementing patterns like the DAO pattern or by introducing new design artefacts that guide us in building the table structures such as the relational model that is

transformed from the domain model. These concepts will unquestionably help us in achieving better data organisation knowledge and will help us improve our skills overall.

Introduction

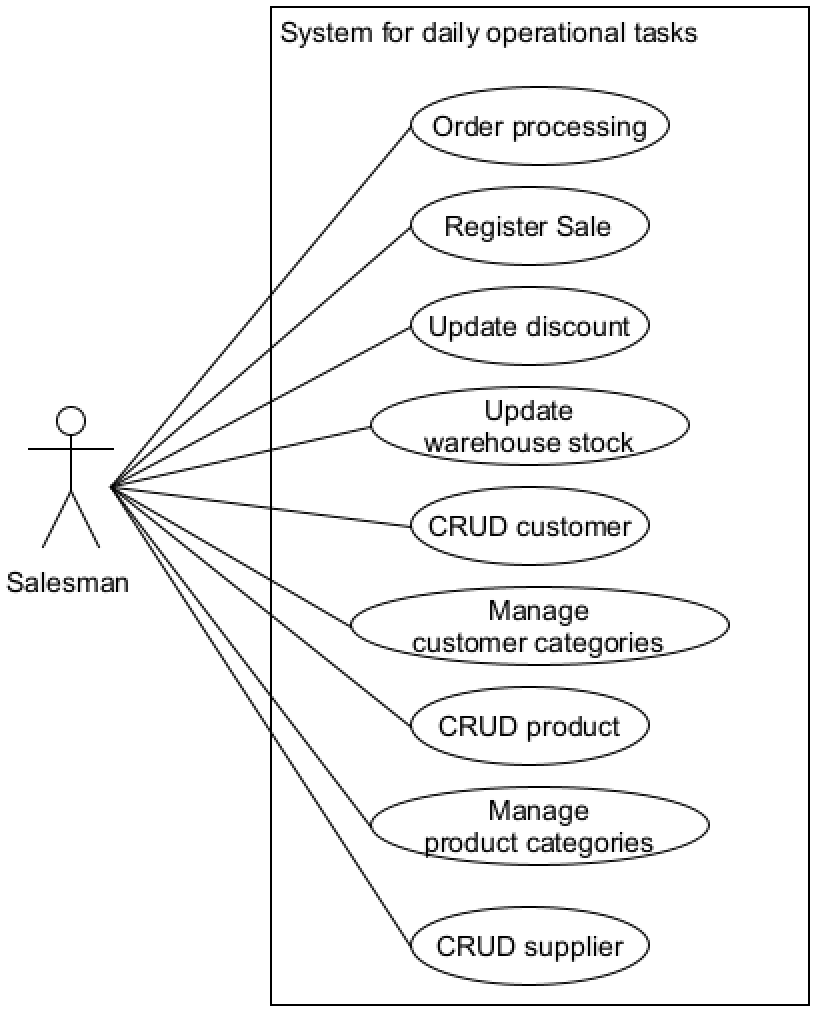
When we started the project, the main focus was based on the understanding of the requirements and somewhat grasping of all the vital details that were mandatory for constructing the right software solution. We prioritised the use cases as the project case was demanding and we started working on the iterations, taking each use case into consideration, one at a time. That allowed us to be more organised and to be able to deliver the asked functionalities in the order of their respective importance. The plan that we made stated that after finishing the designing of the artefacts we could move on to coding the system, starting from the bottom layer and going up to the graphical user interface and implementing tests in the meantime to ensure that our code was generating the expected results.

Scope of the project

The scope of the project is to successfully implement the system that was contouring out of the requirements given in the project case. The main parts that we approached in this project were the concepts of system designing and the implementation of a three-layered architecture solution, in which we managed to apply all the previously learned techniques and principles, including the essential database for data storage and the test package for ensuring the functionality and the quality of the software product.

Use case diagram

Based on the summary of the project case we devised this use case diagram. The scope of the project is to cover the business activities of the salesman. These activities are like processing an order or registering a sale and registering customers or products. In the text there are mentioned more processes, but they are beyond the scope of our project. These activities are like budgeting, accounting tasks, managing procurements and keeping track of stock in different warehouses in case the company wants to expand. (Knoll knol, n.d.)



Use case diagram

Fully-dressed use case - Order processing

Order processing use case is the most important use case and the one we started working on at first. This is why we are focusing on it during the first iteration. We prepared for design by creating a Fully-dressed use case. Our happy scenario happens when the customer sends an email and calls the salesman to order some products. In the case it was not said the customer can buy items in place therefore we're not implementing scanners. Salesman searches for the customer, choses products to sell and then finishes the order, then invoice is sent.

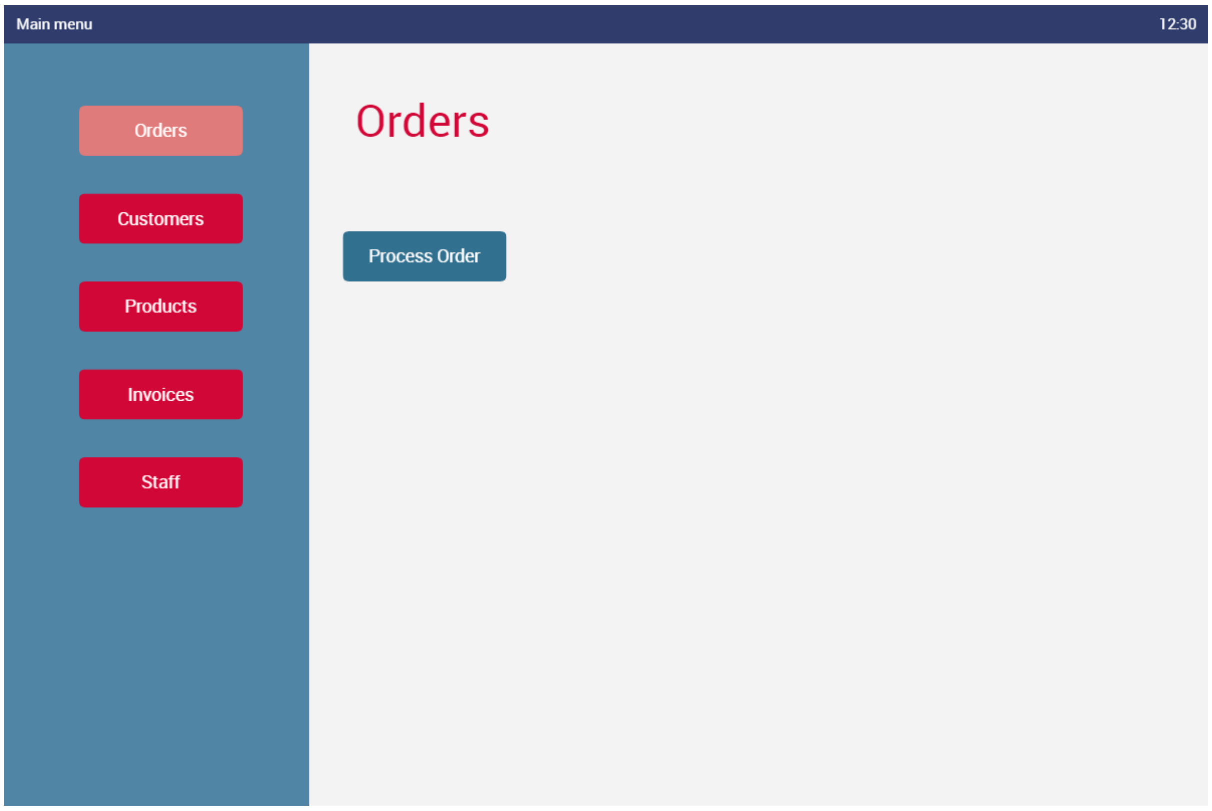
Use case name	**Order processing**
Actors	**Salesman**

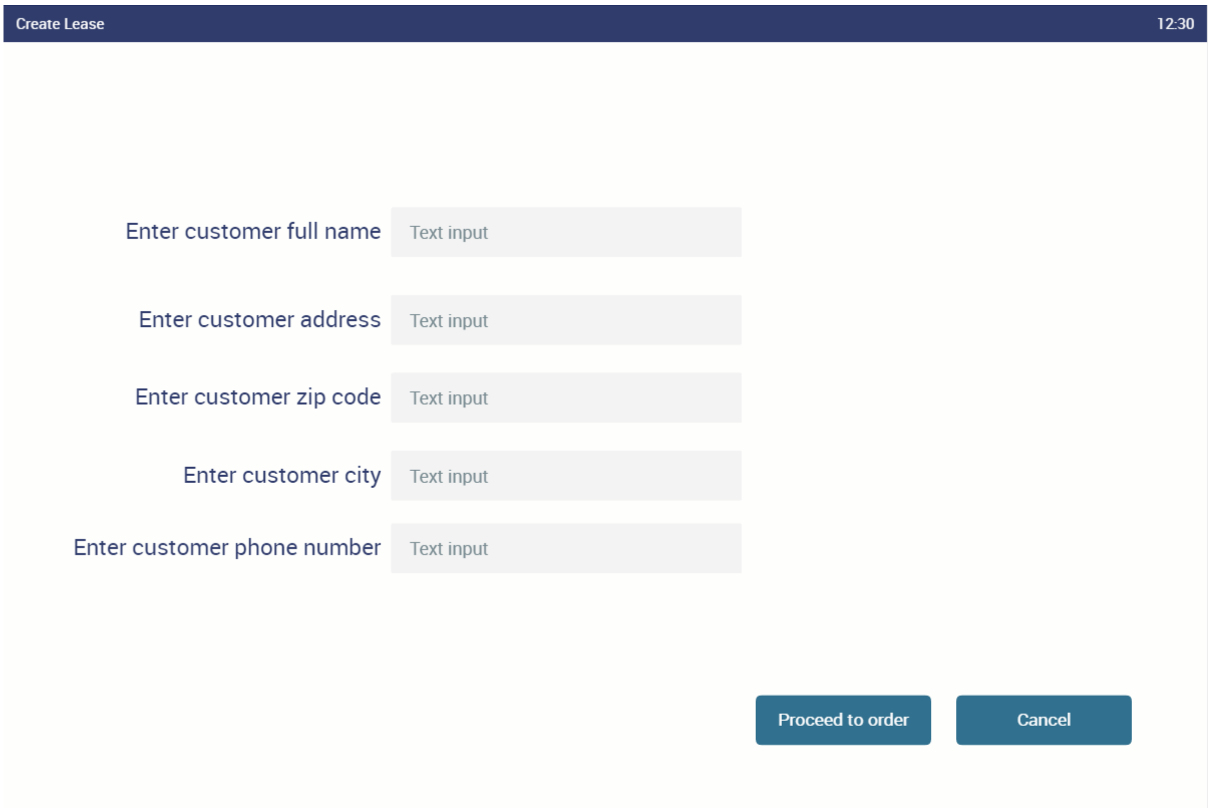
Pre-conditions	**Product and the Customer exists**	
Post-conditions	**Sale order is created and invoice is sent**	
Frequency	**Every time a customer wants to buy product**	
Main Success Scenario	**Actor (Action)**	**System (Response)**
	1. Salesman searches for registered customer	**2. System finds a customer and returns response**
	3. Salesman choses a product and inserts amount	**4. System check if the products are available in the stock**
		5. System adds products to the order line
	Repeat step 3,4 and 5 until all products have been scanned	
	6. Salesman confirms the order	**7. System calculates the price and adds the price of delivery if needed**
		8. System prints confirmation and sends invoice
Alternative flows	***2a. Customer is not in the system*** ***1. The salesman creates a new customer*** ***2. The salesman proceeds*** ***4a. System cannot find a product*** ***System asks to retry the search*** ***5a. Order is cancelled*** ***System comes back to the main menu***	

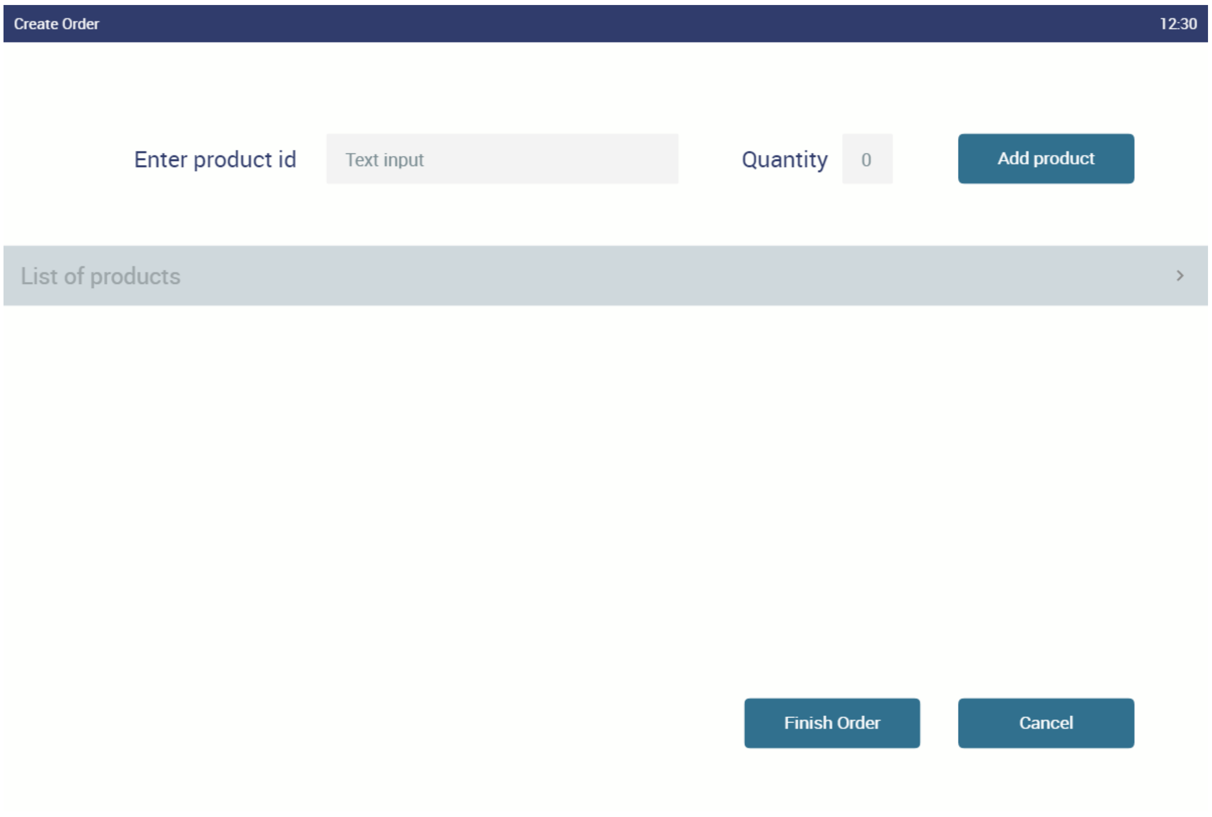
Fully-dressed use case

Mock-ups

Before we started to design and code our system we needed an idea of how the gui design will look like. For that we have used an online tool called FluidUI. In the images below we only showed a few important windows that imaged our vision about the system design. The buttons that are on the side of the first window describe the main functionalities that we discovered, whilst the last two are focused on the registration of a customer and the placing of the order.


Main menu
12:30
Orders
Customers
Products
Invoices
Staff
Orders
Process Order

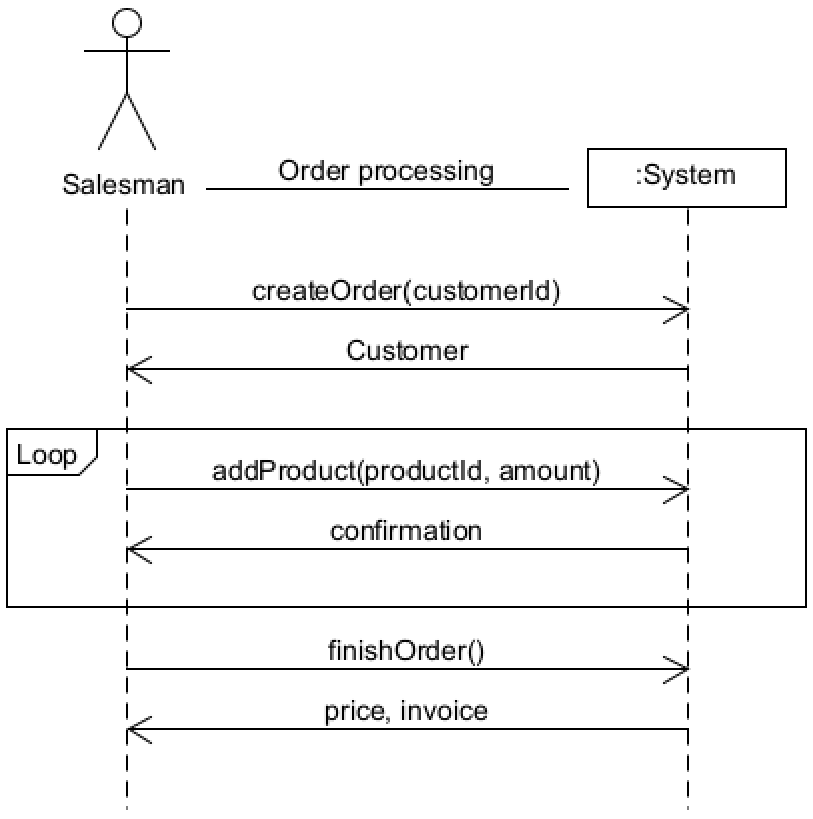

Create Lease
12:30
Enter customer full name
Text input
Enter customer address
Text input
Enter customer zip code
Text input
Enter customer city
Text input
Enter customer phone number
Text input
Proceed to order
Cancel



Mockups

System sequence diagram

Based on the fully-dressed use case we created system sequence diagrams. It represents the interactions between the user and the system which are the actors of the use case.



SSD

Operation contracts

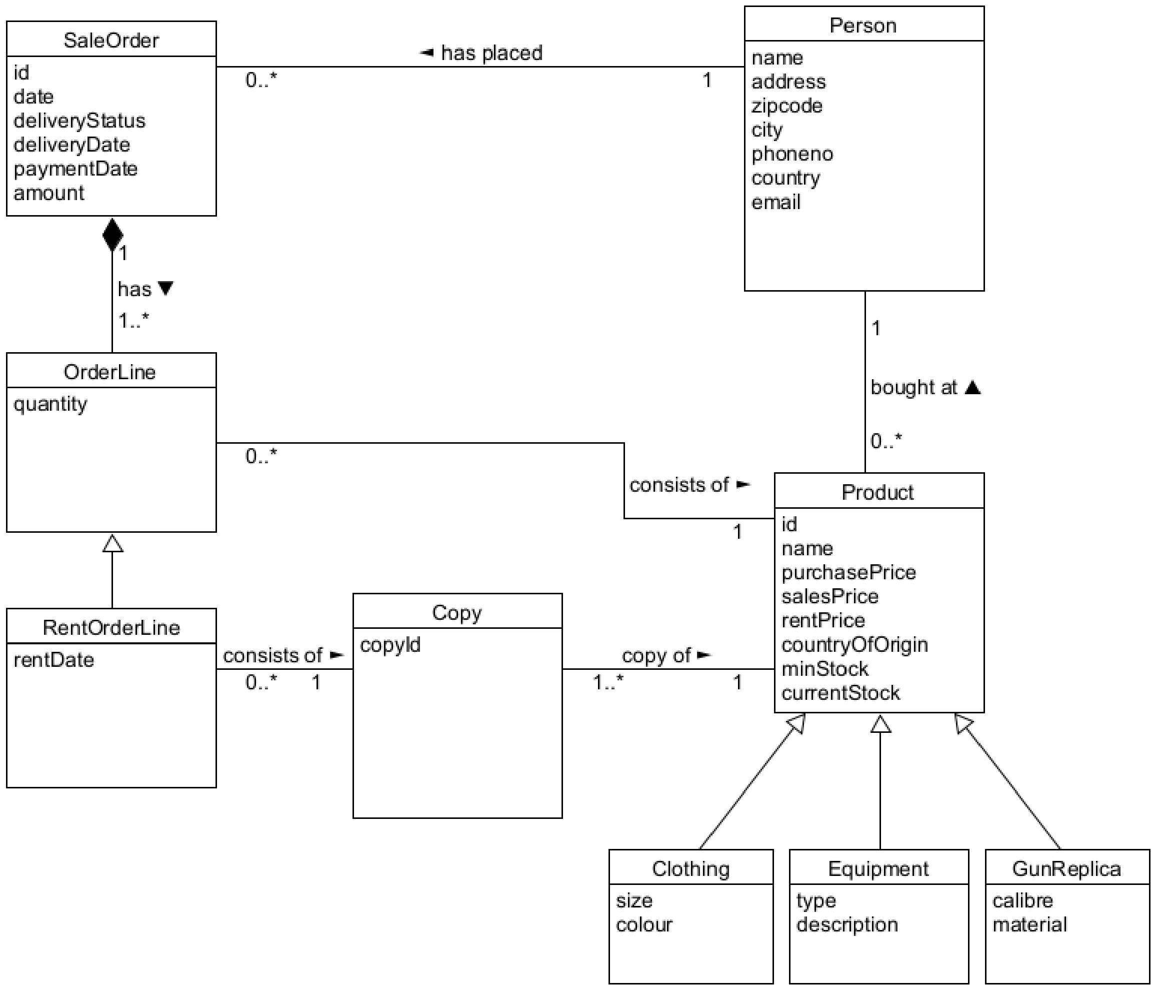
This operation contract relates to the method that changes something in the system. addProduct() adds the product in a saleOrderLine into the saleOrder.

Operation	addProduct()
Use Case	Order processing
Precondition	Customer c, Product p, Order o exists
Postcondition	● s(saleOrderLine) is created ● p is associated with s ● p amount is changed ● s is added to o

Operation contract

Domain model

We have added an order line to the sale order, so the customer can order more product types. And also an id to the product.



Domain model

Relational model

After finishing the domain model we wanted to create the relational model for the databases too. For this first we copied the classes from the domain model as the different tables. For the generalisation with the product we chose to create tables for each class instead of using the pull-up or pull-down methods. We did not choose the pull-up because in the future there can be easily a need for the system to be extended and doing that with the pull-up is quite difficult since we have to extend the whole table, in the meantime we also avoid the problems with the NULL values. We opted out of using the pull-down because the superclass has many attributes and implementing all of them in all subclasses would have been a waste of resources. So instead we created a table for each class, this way only the searching will be difficult since we have to use a join when doing so.
But with the other generalisation with the order lines we used the pull-down method, because in the superclass we did not have a candidate class that could identify that class so we did not want to add a new attribute for identifications.

Person						
name	address	country	zipcode	city	phoneno	email

SaleOrder					
id	date	deliveryStatus	deliveryDate	paymentDate	amount

SaleOrderLine	
quantity	saleId

RentOrderLine	
rentDate	quantity

Copy		
copyId	isRentable	rentDate

Product							
id	name	purchasePrice	salesPrice	rentPrice	countryOfOrigin	minStock	currentStock

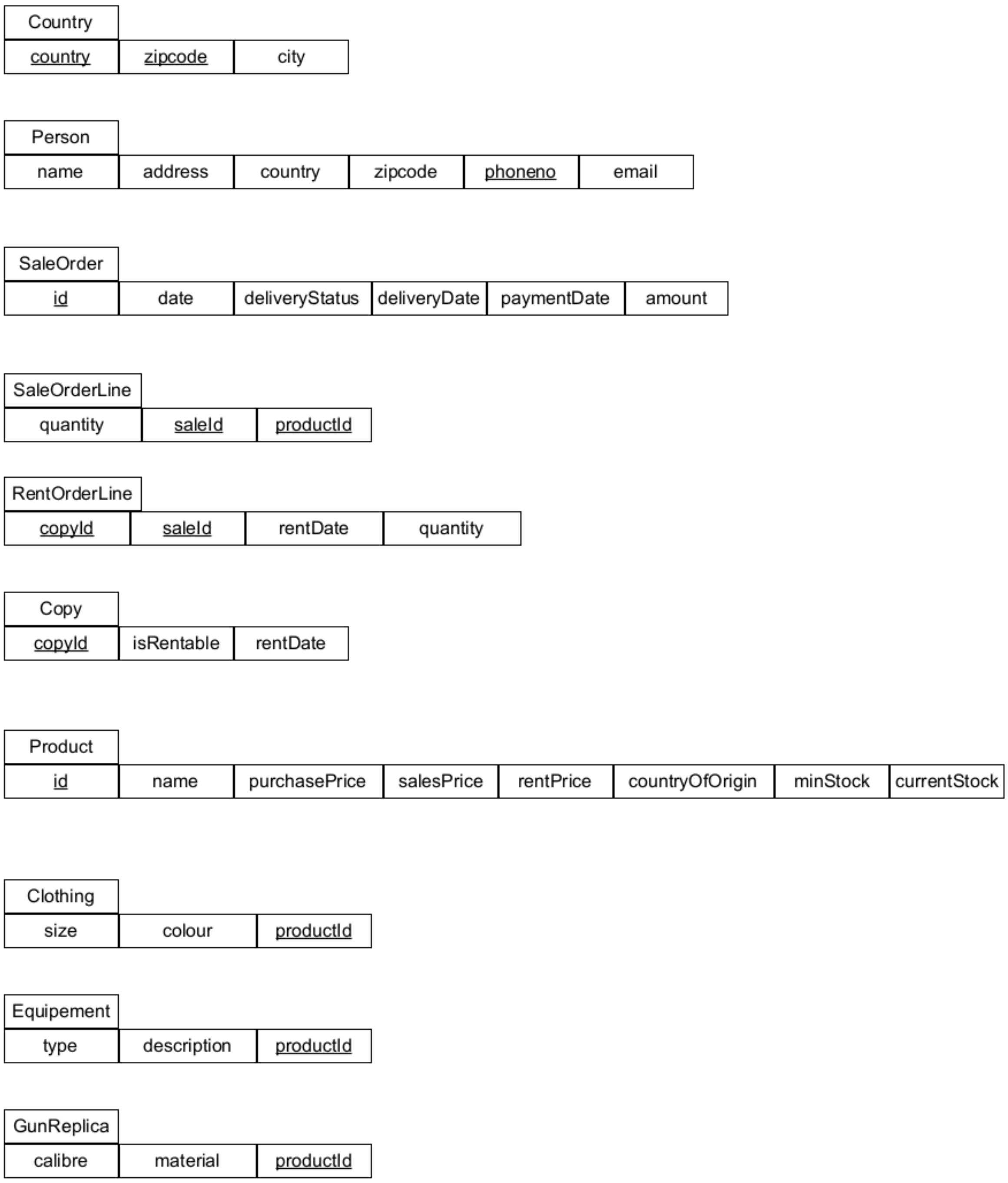
Clothing	
size	colour

Equipement	
type	description

GunReplica	
calibre	material

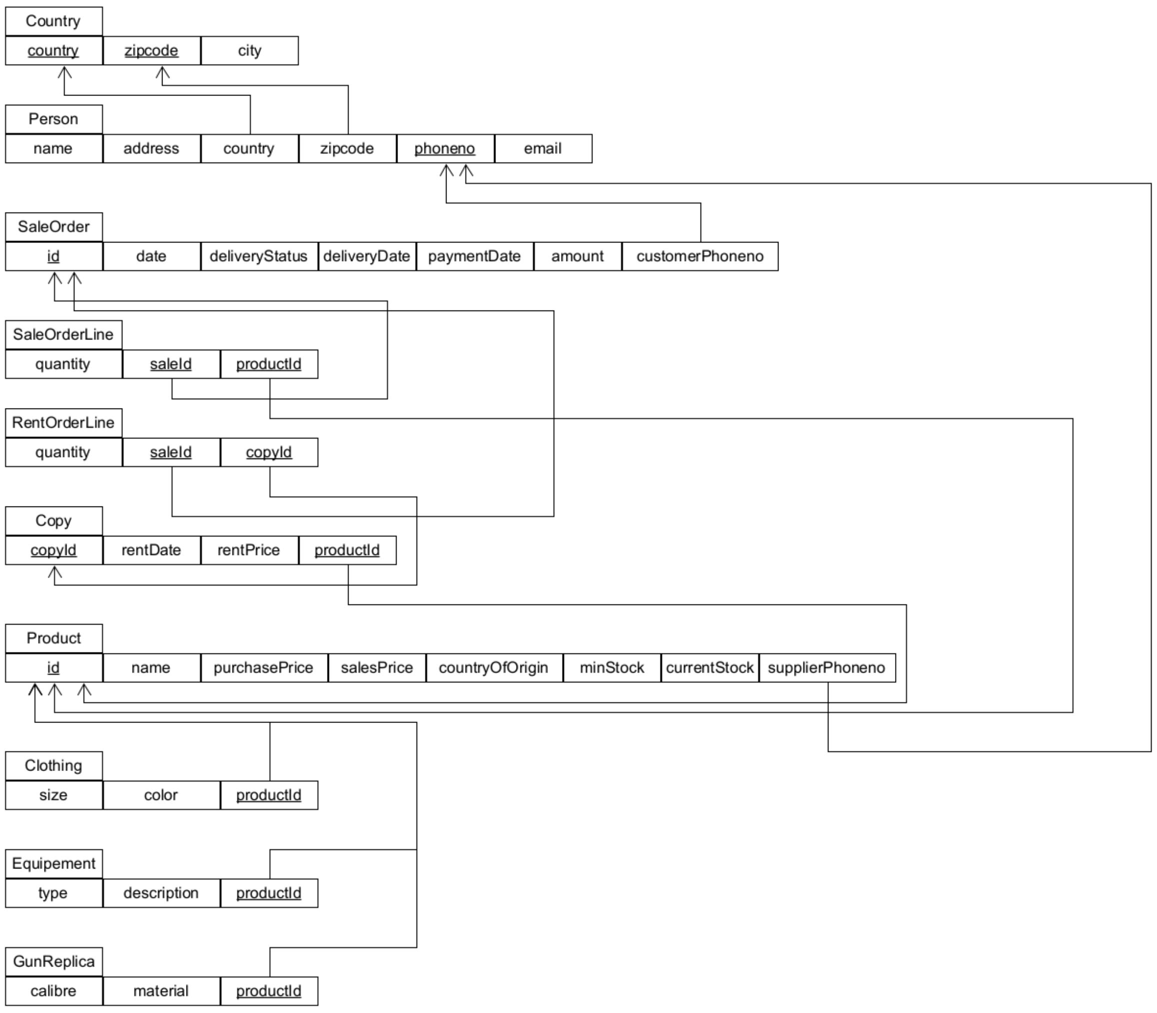
Relational model 1

After having the classes and attributes, we choose one attribute in each class to be the primary key. These attributes are the ones which are identifying the different tuples and making sure that we do not have any tuple with all the same data, because there can be no two tuples with the same in the primary key. In the classes where we did not have the necessary attributes we created new ones, in some cases keeping in mind what foregin keys we were planning to use and choosing the primary keys from those.
We also realised that the zip code has a reference to the city, so to follow the 3rd normalisation form



Relational model 2

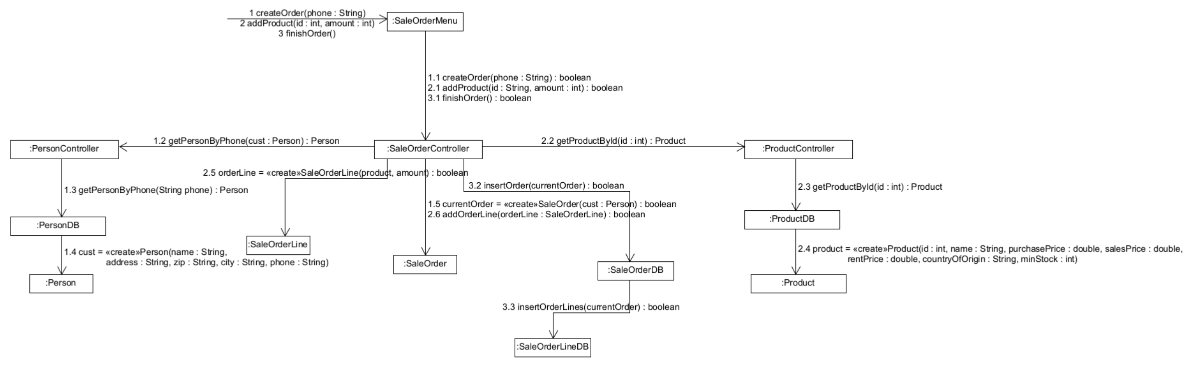
Lastly we set up all the foreign keys with all the different attributes that they are referencing to and made some modifications to the attributes. After that we used this final diagram to create the different tables in the database and inserted data into them.



Relational model 3

Interaction diagram

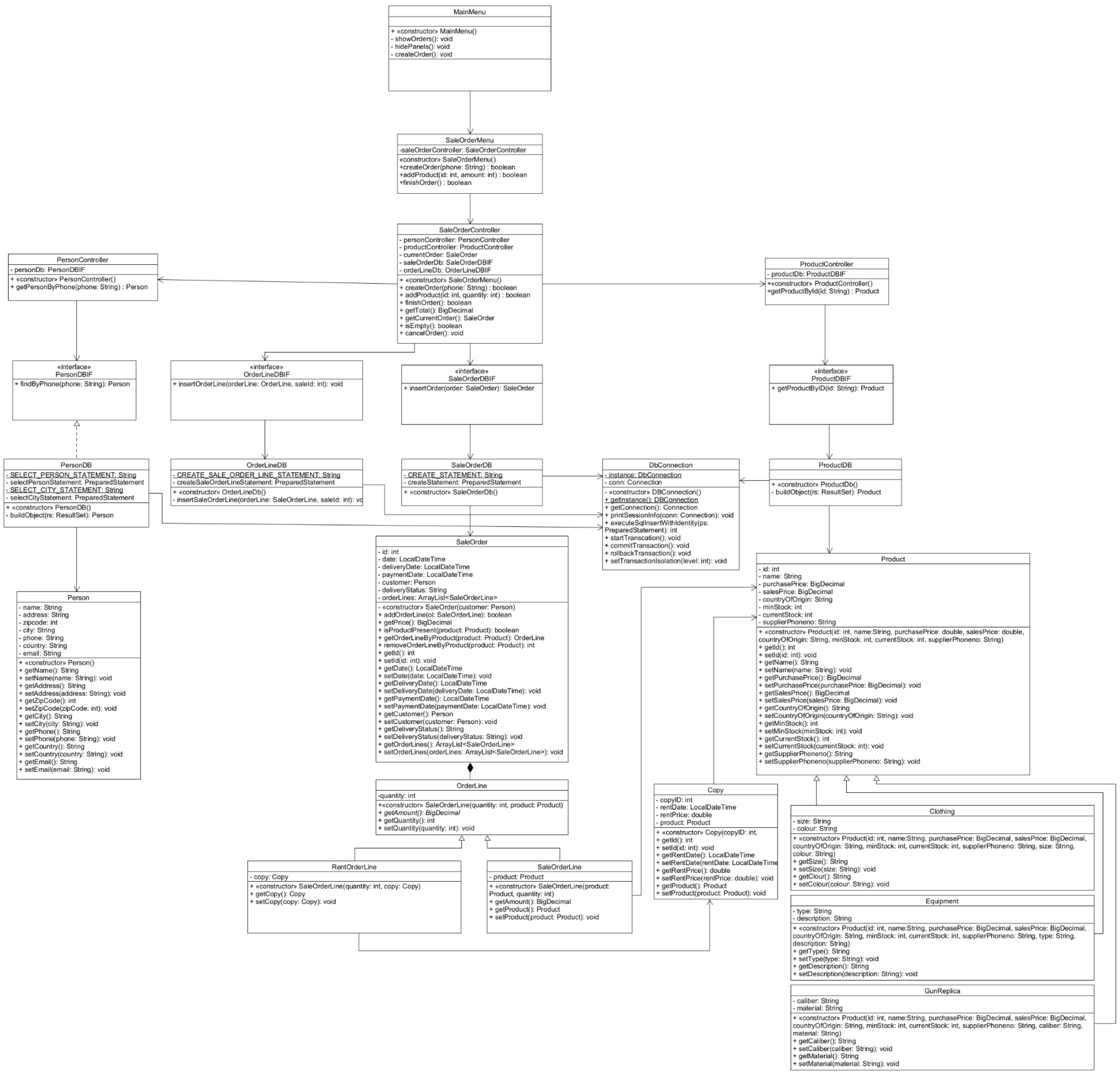
Interaction diagram shows what methods are called on which classes with order. For use case order processing the user is calling three methods on the menu. The menu will forward the information to the controllers. The controller asks the model layer for the objects. The model layer consists of a database access layer and classes representing the database data. The representing classes are retrieved to the controller which are used for the order. In this case it is the customer and the products.


1 createOrder(phone : String)
2 addProduct(id : int, amount : int)
3 finishOrder()
:SaleOrderMenu
1.1 createOrder(phone : String) : boolean
2.1 addProduct(id : String, amount : int) : boolean
3.1 finishOrder() : boolean
:PersonController
1.2 getPersonByPhone(cust : Person) : Person
:SaleOrderController
2.2 getProductById(id : int) : Product
:ProductController
2.5 orderLine = «create»SaleOrderLine(product, amount) : boolean
3.2 insertOrder(currentOrder) : boolean
1.3 getPersonByPhone(String phone) : Person
1.5 currentOrder = «create»SaleOrder(cust : Person) : boolean
2.6 addOrderLine(orderLine : SaleOrderLine) : boolean
2.3 getProductById(id : int) : Product
:PersonDB
:ProductDB
:SaleOrderLine
1.4 cust = «create»Person(name : String, address : String, zip : String, city : String, phone : String)
:SaleOrder
2.4 product = «create»Product(id : int, name : String, purchasePrice : double, salesPrice : double, rentPrice : double, countryOfOrigin : String, minStock : int)
:SaleOrderDB
:Person
:Product
3.3 insertOrderLines(currentOrder) : boolean
:SaleOrderLineDB

Communication diagram

Design class diagram

Our design class diagram is designed in three layers. We differentiate the UI layer, which takes care of the interactions with the user. Controller layer, that is responsible for logic of the system. Finally we have a model layer and database layer, both at the same level of the system, responsible for storing the data, inputing data to the database and restoring it from the database into an instance of objects from the system. Here we aimed for open architecture that, with the amount of time we were given, allowed us to focus more on quality of the database implementation (as it was in the focus of the project).

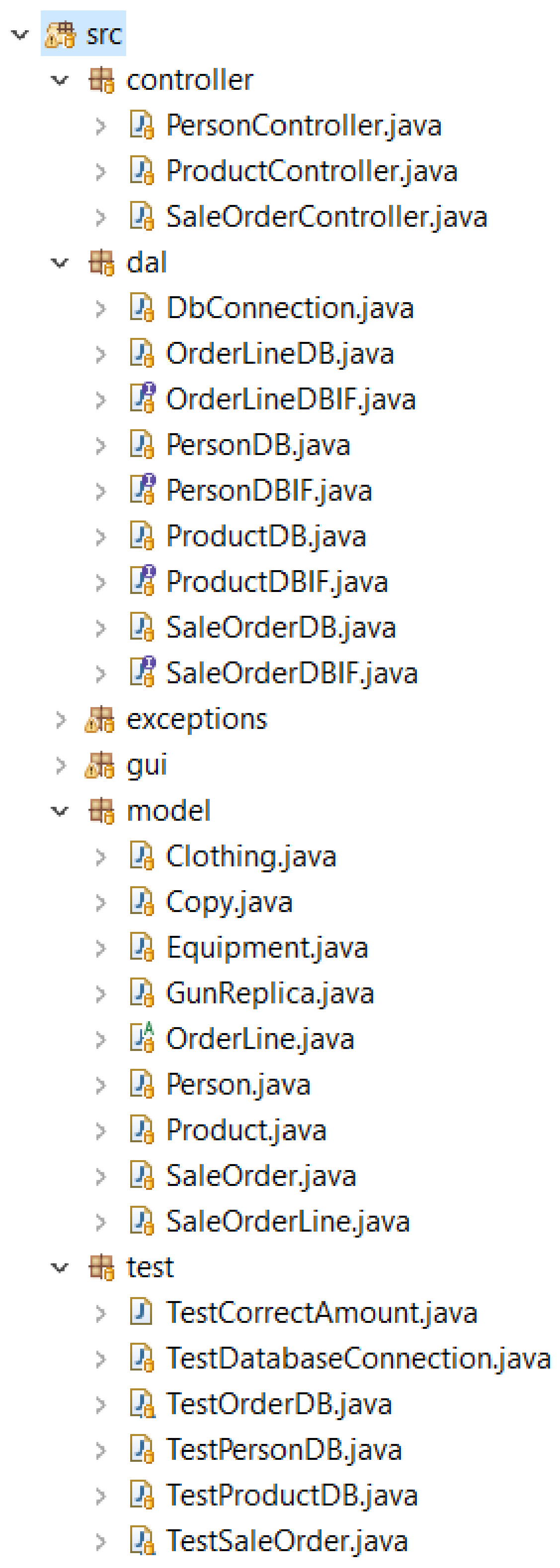


Design class diagram

Architecture

Data access object

This structural pattern helps the programmers to isolate the business layer from the persistence layer using an abstract API. In the picture below you can see our packages and how we separated our layers. We differentiate the controller layer, model layer, persistence layer (this was previously called container layer). We don't store business information anymore in the application, but in our separated SQL database. In the persistence layer the program handles the commands, which we want to run to execute on the SQL database. See below the architecture of the application. (*The DAO Pattern in Java | Baeldung*, n.d.)



```
src
  controller
    PersonController.java
    ProductController.java
    SaleOrderController.java
  dal
    DbConnection.java
    OrderLineDB.java
    OrderLineDBIF.java
    PersonDB.java
    PersonDBIF.java
    ProductDB.java
    ProductDBIF.java
    SaleOrderDB.java
    SaleOrderDBIF.java
  exceptions
  gui
  model
    Clothing.java
    Copy.java
    Equipment.java
    GunReplica.java
    OrderLine.java
    Person.java
    Product.java
    SaleOrder.java
    SaleOrderLine.java
  test
    TestCorrectAmount.java
    TestDatabaseConnection.java
    TestOrderDB.java
    TestPersonDB.java
    TestProductDB.java
    TestSaleOrder.java
```

Arhitecture

Database layer

We have an interface for every class that implements connection to the database. This is useful in the case, when the database is changed, because we can just change the implementation of the database access and nothing else.

```java
package dal;

import exceptions.DatabaseAccessException;

public interface SaleOrderDBIF {

    public SaleOrder insertOrder(SaleOrder order) throws DatabaseAccessException;

}
```

Figure 13.

In the implementation of the interface we made a constant value of the statement that is initialised as a prepared statement in the constructor, which prevents sql injection.

```java
package dal;

import java.sql.PreparedStatement;

public class SaleOrderDB implements SaleOrderDBIF {

    private static final String CREATE_STATEMENT = "INSERT INTO SaleOrder(date, deliveryStatus, deliveryDate,"
            + " paymentDate, amount, customerPhoneno) VALUES(?, ?, ?, ?, ?, ?)";

    private PreparedStatement createStatement;

    public SaleOrderDB() throws DatabaseAccessException {
        try {
            createStatement = DbConnection.getInstance().getConnection().prepareStatement(CREATE_STATEMENT,
                    Statement.RETURN_GENERATED_KEYS);
        } catch (SQLException e) {
            //e.printStackTrace();
            throw new DatabaseAccessException(DatabaseAccessException.CONNECTION_MESSAGE);
        }
    }
```

Figure 14.

In the method we add the values to the prepared statement and then execute it.

```java
/**
 * Inserts an order into the database and sets the order's id to the one generated by the database
 * @param order the order to be inserted into the database
 * @return the order with the set id
 * @throws DatabaseAccessException
 */
@Override
public SaleOrder insertOrder(SaleOrder order) throws DatabaseAccessException {
    try {
        createStatement.setTimestamp(1, Timestamp.valueOf(order.getDate()));
        createStatement.setString(2, order.getDeliveryStatus());
        createStatement.setTimestamp(3, Timestamp.valueOf(order.getDeliveryDate()));
        createStatement.setTimestamp(4, Timestamp.valueOf(order.getPaymentDate()));
        createStatement.setBigDecimal(5, order.getPrice());
        createStatement.setString(6, order.getCustomer().getPhoneno());

        order.setId(DbConnection.getInstance().executeSqlInsertWithIdentity(createStatement));

    } catch (SQLException e) {
        //e.printStackTrace();
        throw new DatabaseAccessException(e.getMessage());
    }

    return order;
}
```

Database layer

Tests

The basic flow of events:

1. Check if the customer is valid.
2. Display the information of the customer.
3. Display the add product button and cancel.
4. Check if the product is valid.
5. Display the information of the product.
6. Display the create order button and cancel.
7. Registers the order in the database.
8. Display the price of the order.
9. Display ready for next order.

The alternative flow of events

1. The customer ID is invalid.
2. The product ID is invalid.
3. The order processing is cancelled.

Use case scenarios for *Order processing*.
These are the possible scenarios for the order processing use case. We create this table from the fully dressed use case.

Scenario 1	Order is processed	Basic flow	
Scenario 2	The customer is new	Alternate flow	A1
Scenario 3	The product doesn't exist	Alternate flow	A2
Scenario 4	Process is cancelled	Alternate flow	A3

Use case scenarios

Testing for valid and invalid values. V stands for Valid, I stands for Invalid.

Test Case ID	Scenario	Customer	Product	Confirm process	Expected result
R1	Scenario 1- Order is processed	V	V	V	The Order is placed for a customer.
R2	Scenario 2 - The customer is new	I	n/a	n/a	Request for creating new customer
R3	Scenario 3 - The product doesn't exist	V	I	n/a	Request for a product
R4	Scenario 4 - Process is cancelled	V	V	I	Process cancelled

Test cases without values

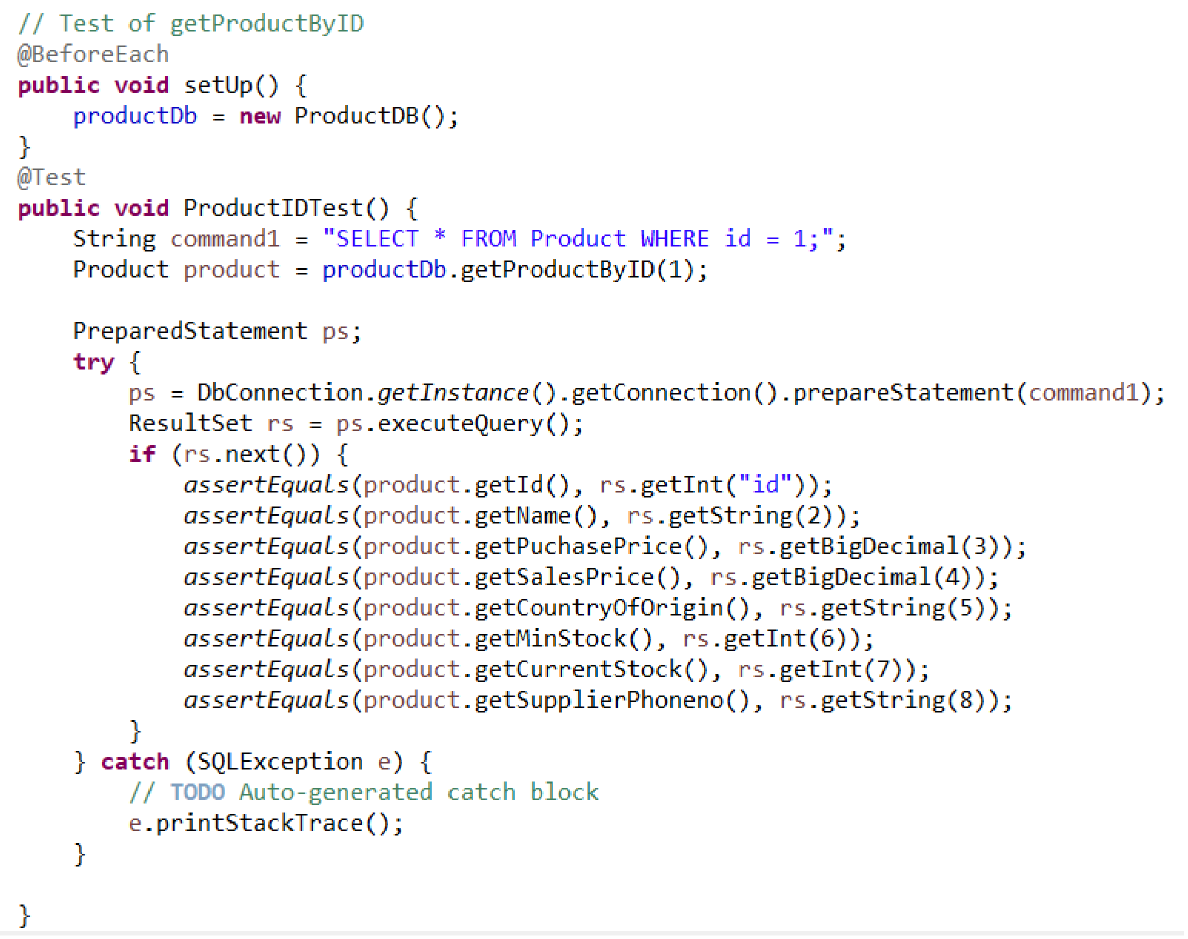
Testing with real values

Test Case ID	Scenario	Customer	Product	Confirm process	Expected result
R1	Scenario 1- Order is processed	Joe (existing customer)	AK-47	Confirm	The Order is placed for a customer.
R2	Scenario 2 - The customer is new	James (not existing customer)	n/a	n/a	Request for creating new customer
R3	Scenario 3 - The product doesn’t exist	Joe (existing customer)	Cheese cake	n/a	Request for a product
R4	Scenario 4 - Process is cancelled	James (existing customer)	AK-47	Cancel	Process cancelled

Test cases with values

Integration tests

We did integration tests to verify if the connection between the classes are operable. This method is useful to test software modules as a group.
In TestProductDB we test the integration ProductDB, DBConnection and Product class. The software checks if the result of our getters match with the ones from the SQL database. See below an example of how we managed to do this task.



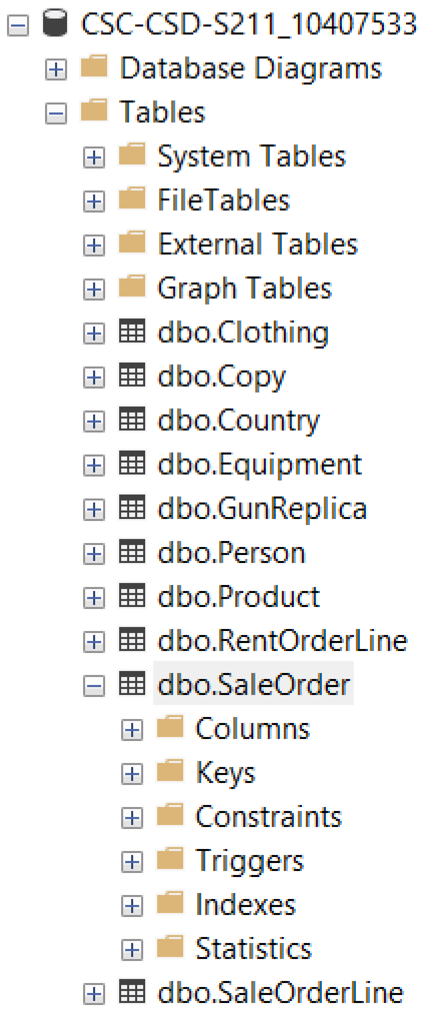
```
// Test of getProductByID
@BeforeEach
public void setUp() {
    productDb = new ProductDB();
}
@Test
public void ProductIDTest() {
    String command1 = "SELECT * FROM Product WHERE id = 1;";
    Product product = productDb.getProductByID(1);

    PreparedStatement ps;
    try {
        ps = DbConnection.getInstance().getConnection().prepareStatement(command1);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            assertEquals(product.getId(), rs.getInt("id"));
            assertEquals(product.getName(), rs.getString(2));
            assertEquals(product.getPuchasePrice(), rs.getBigDecimal(3));
            assertEquals(product.getSalesPrice(), rs.getBigDecimal(4));
            assertEquals(product.getCountryOfOrigin(), rs.getString(5));
            assertEquals(product.getMinStock(), rs.getInt(6));
            assertEquals(product.getCurrentStock(), rs.getInt(7));
            assertEquals(product.getSupplierPhoneno(), rs.getString(8));
        }
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

Integration tests

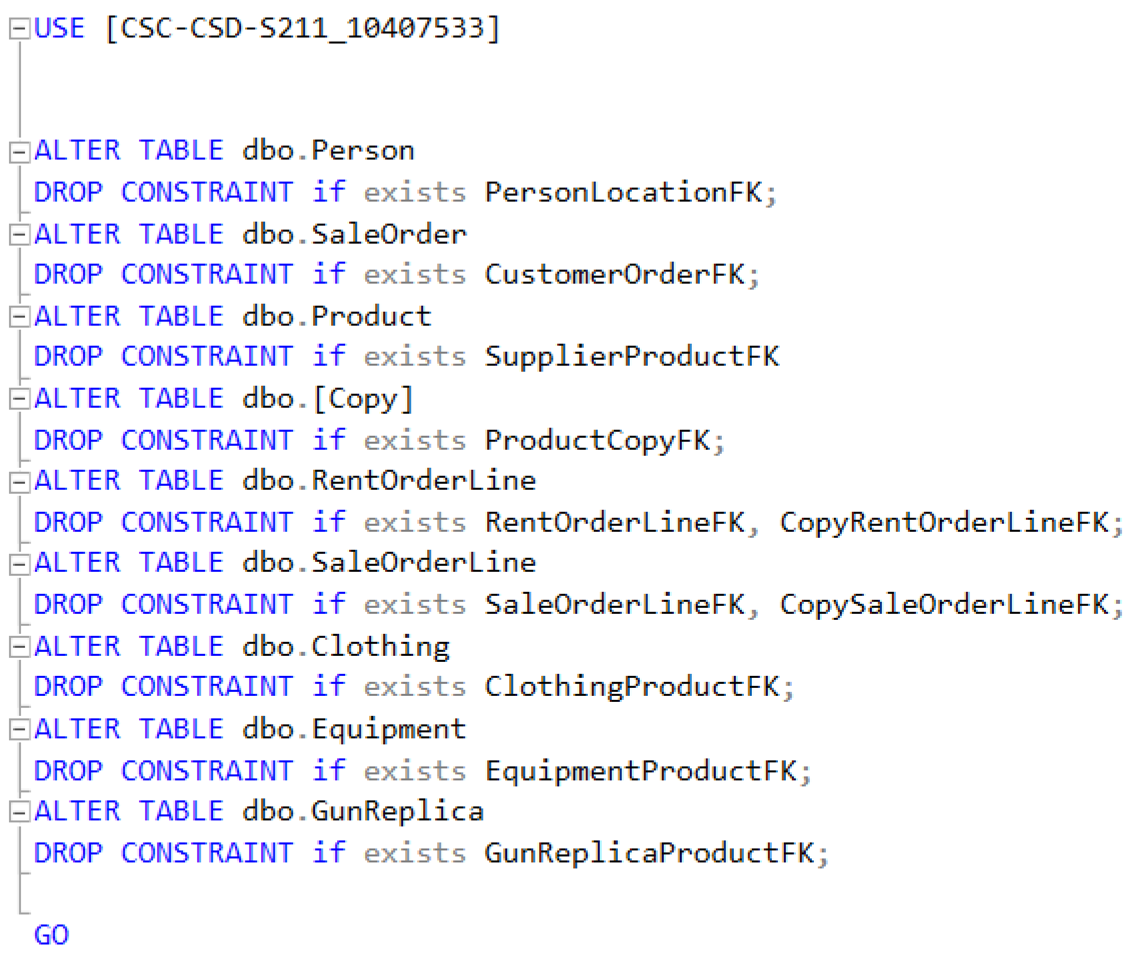
In TestSaleOrder we check if the multiplication works with hard coded values. We execute this multiplication by placing one variable in orderLine and placing the other in saleOrderLine. If the multiplication returns the desired value, then the integration is successful.
In TestPersonDB integration PersonDB, DBConnection and Person class. The software checks if the result of our getters match with the ones from the SQL database.
In TestOrderDB integration OrderDB, SaleOrderLine, DBConnection, ProductDB and PersonDB class. The software checks if the result of our getters match with the ones from the SQL database.
TestDatabaseConnection is for checking the successfulness of the connection. The test makes sure if it can get an instance of the DbConnection. If not null, then it was successful.

SQL scripts and tables

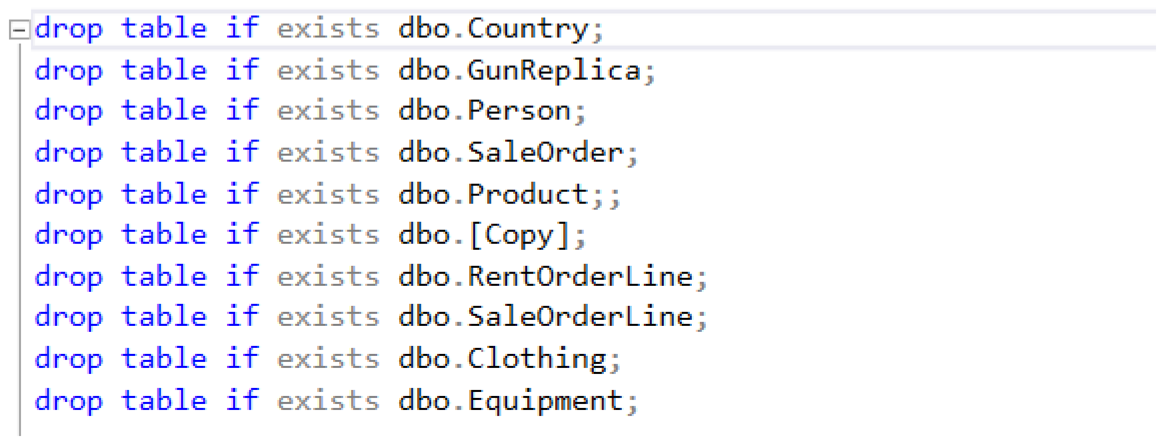
The image below shows an overview of how the database looked after the creation of the tables. We only used one database, from a person from the group, for testing our sql scripts.



Every time something didn’t work out, or when we simply inserted the data into the tables we had to drop them first and then recreate them from scratch. The code down below shows how we are first altering the table to get rid of the constraint and then, if the table exists, dropping it. This way we are ensuring that there are no errors while trying to drop the tables, because there is no foreign key connected between them anymore.

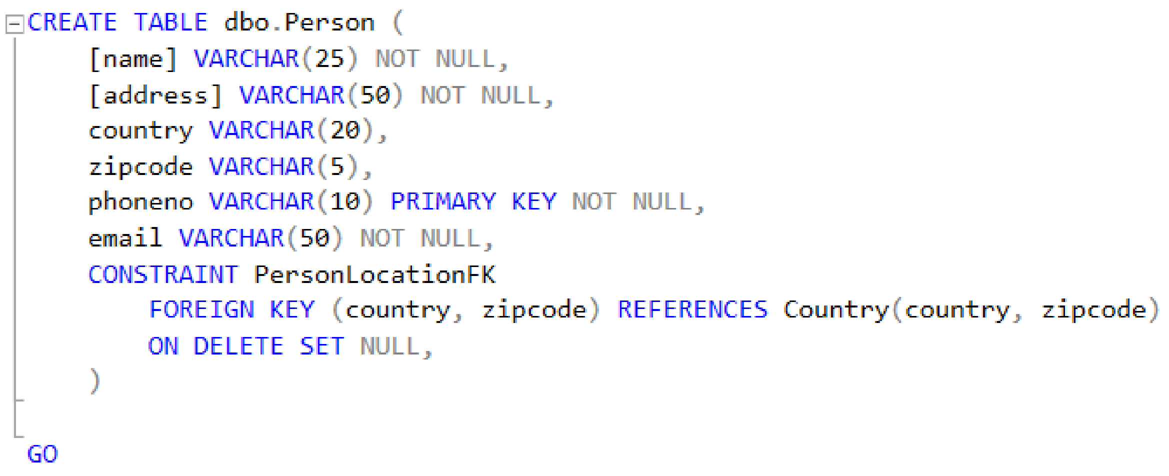


```sql
USE [CSC-CSD-S211_10407533]


ALTER TABLE dbo.Person
DROP CONSTRAINT if exists PersonLocationFK;
ALTER TABLE dbo.SaleOrder
DROP CONSTRAINT if exists CustomerOrderFK;
ALTER TABLE dbo.Product
DROP CONSTRAINT if exists SupplierProductFK
ALTER TABLE dbo.[Copy]
DROP CONSTRAINT if exists ProductCopyFK;
ALTER TABLE dbo.RentOrderLine
DROP CONSTRAINT if exists RentOrderLineFK, CopyRentOrderLineFK;
ALTER TABLE dbo.SaleOrderLine
DROP CONSTRAINT if exists SaleOrderLineFK, CopySaleOrderLineFK;
ALTER TABLE dbo.Clothing
DROP CONSTRAINT if exists ClothingProductFK;
ALTER TABLE dbo.Equipment
DROP CONSTRAINT if exists EquipmentProductFK;
ALTER TABLE dbo.GunReplica
DROP CONSTRAINT if exists GunReplicaProductFK;

GO
```



```sql
drop table if exists dbo.Country;
drop table if exists dbo.GunReplica;
drop table if exists dbo.Person;
drop table if exists dbo.SaleOrder;
drop table if exists dbo.Product;;
drop table if exists dbo.[Copy];
drop table if exists dbo.RentOrderLine;
drop table if exists dbo.SaleOrderLine;
drop table if exists dbo.Clothing;
drop table if exists dbo.Equipment;
```

We created the tables based on the relational model. As the image below shows, we tried to be consistent by using the same amount of columns with the same names and with the right foreign key according to the relational model that was designed.



```
CREATE TABLE dbo.Person (
    [name] VARCHAR(25) NOT NULL,
    [address] VARCHAR(50) NOT NULL,
    country VARCHAR(20),
    zipcode VARCHAR(5),
    phoneno VARCHAR(10) PRIMARY KEY NOT NULL,
    email VARCHAR(50) NOT NULL,
    CONSTRAINT PersonLocationFK
        FOREIGN KEY (country, zipcode) REFERENCES Country(country, zipcode)
        ON DELETE SET NULL,
    )

GO
```

Conclusion

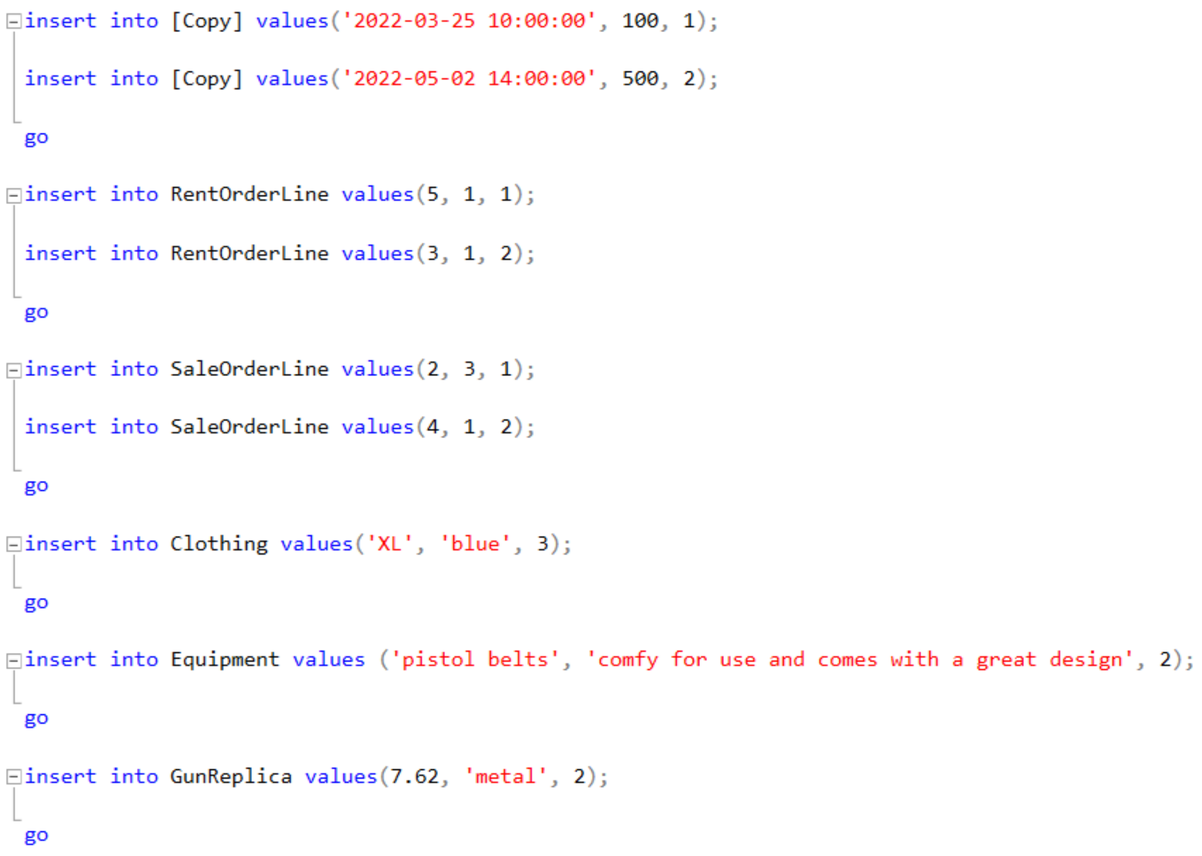
In the end, we realised that the proposed case was suggesting a system that was more complex than we taught in the beginning. Given use case was difficult for us firstly to understand, and then to implement. The biggest challenge we faced was implementing a new way of storing the data and connecting it to a newly discovered type of container, the database. As the system is incomplete, we feel like the tasks given were completed and we all got a way better understanding of database implementation and testing. Given the amount of time we were given we were satisfied with the results of our group work. We managed to fully implement database functionalities, our system's logic is working and we even implemented gui inspired by the first semester project's gui.

References

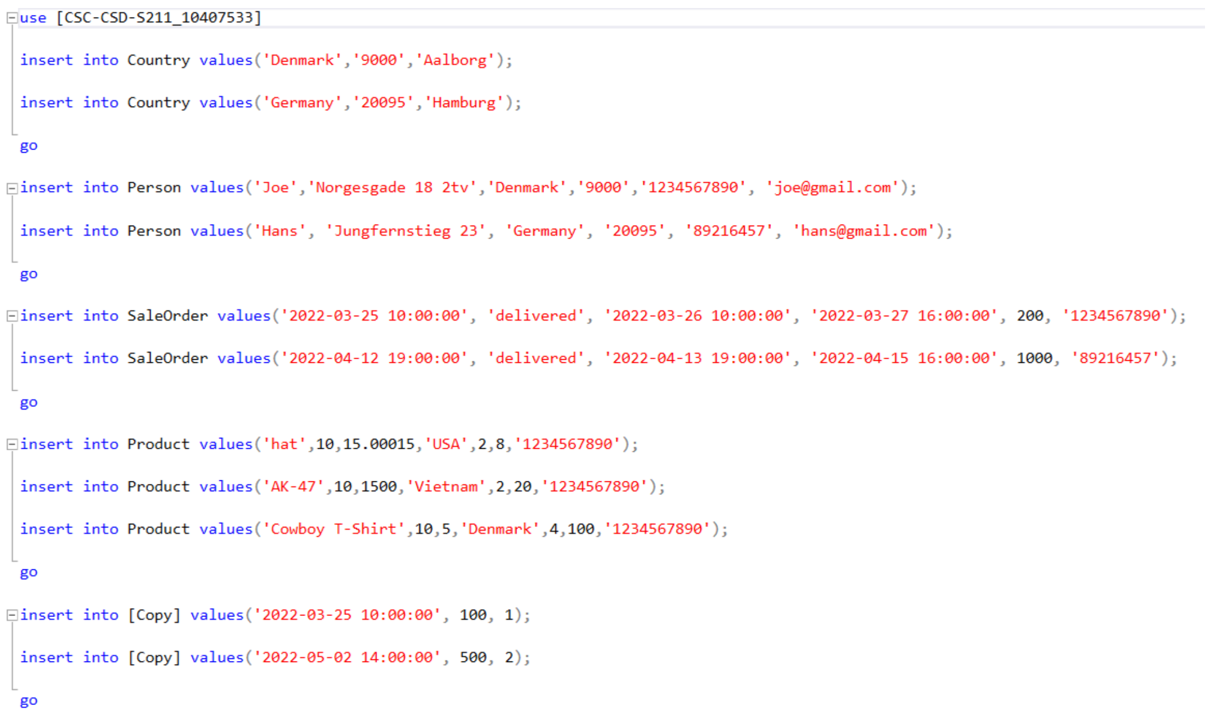
1. *The DAO Pattern in Java | Baeldung*. (n.d.). Retrieved March 25, 2022, from https://www.baeldung.com/java-dao-pattern
2. Knoll knol, I. (n.d.). *A DAO-implementation in Java 1 Lecture Note Persistence An Example Architecture for Encapsulation of Database Access in Java Systems 2nd Edition*.
3. *Larman, Craig, 2005. Applying UML And Patterns. 3rd ed. Upper Saddle River, New Jersey: Prentice-Hall. [Accessed 2022 Mar. 25].*

Appendices

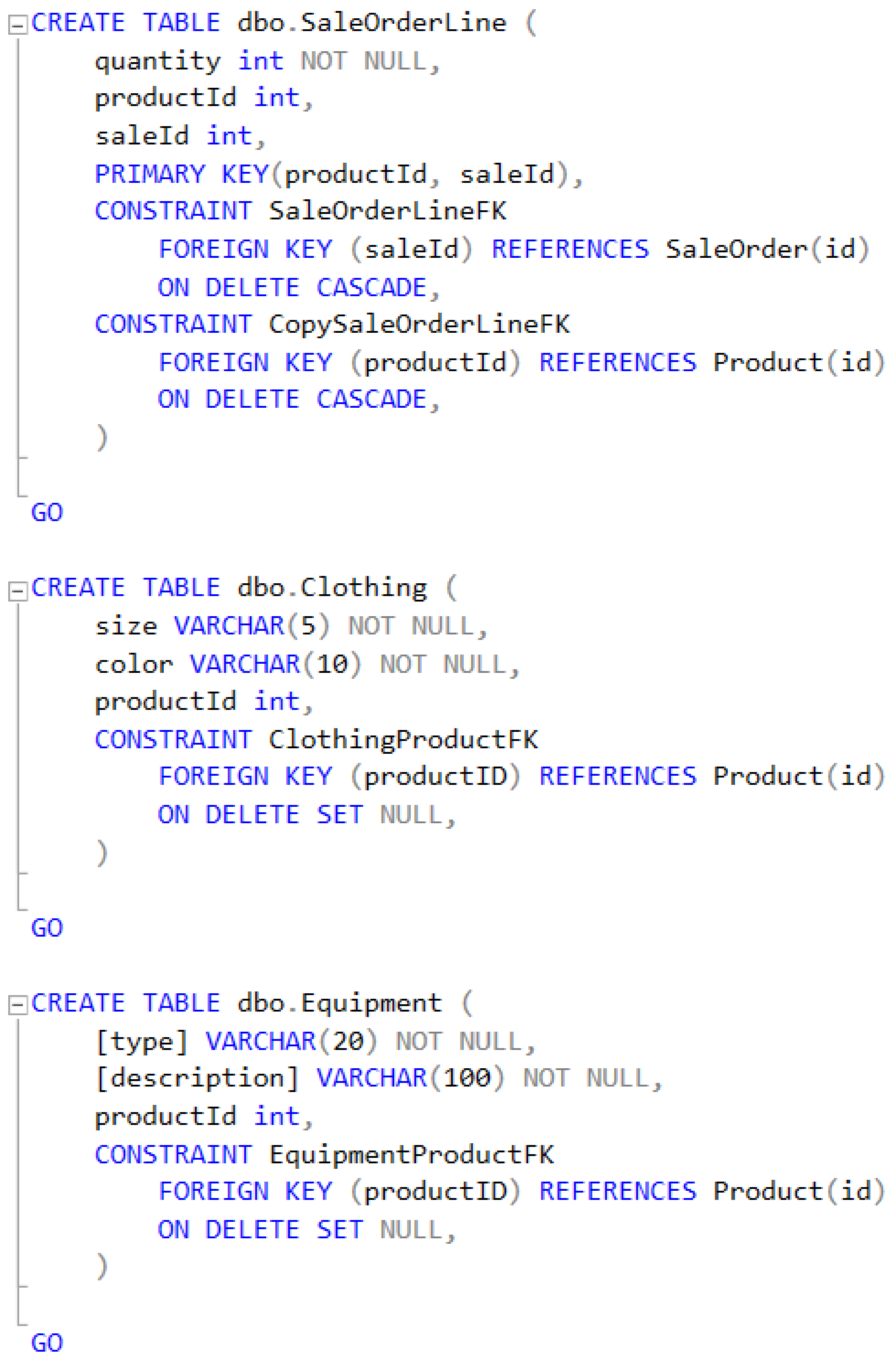
Appendix1



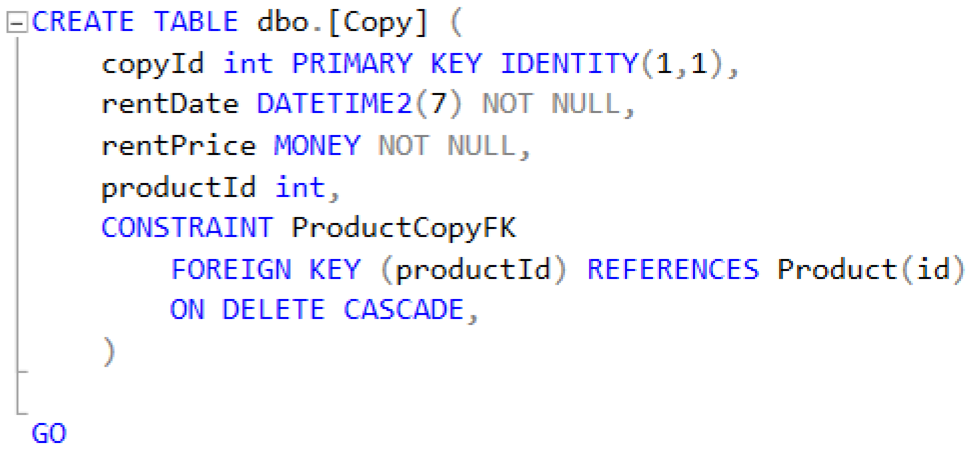
```sql
insert into [Copy] values('2022-03-25 10:00:00', 100, 1);

insert into [Copy] values('2022-05-02 14:00:00', 500, 2);

go

insert into RentOrderLine values(5, 1, 1);

insert into RentOrderLine values(3, 1, 2);

go

insert into SaleOrderLine values(2, 3, 1);

insert into SaleOrderLine values(4, 1, 2);

go

insert into Clothing values('XL', 'blue', 3);

go

insert into Equipment values ('pistol belts', 'comfy for use and comes with a great design', 2);

go

insert into GunReplica values(7.62, 'metal', 2);

go
```



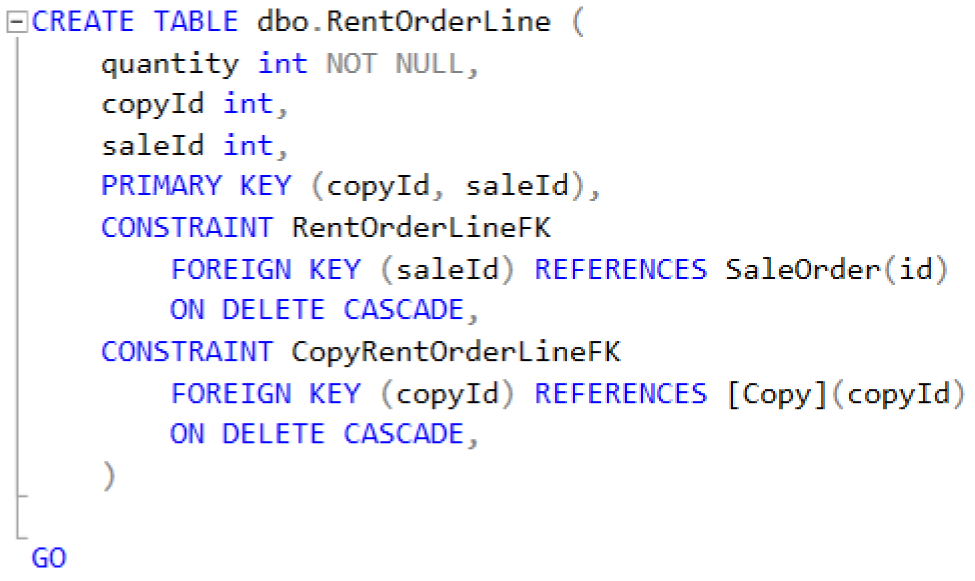
```sql
use [CSC-CSD-S211_10407533]

insert into Country values('Denmark','9000','Aalborg');

insert into Country values('Germany','20095','Hamburg');

go

insert into Person values('Joe','Norgesgade 18 2tv','Denmark','9000','1234567890', 'joe@gmail.com');

insert into Person values('Hans', 'Jungfernstieg 23', 'Germany', '20095', '89216457', 'hans@gmail.com');

go

insert into SaleOrder values('2022-03-25 10:00:00', 'delivered', '2022-03-26 10:00:00', '2022-03-27 16:00:00', 200, '1234567890');

insert into SaleOrder values('2022-04-12 19:00:00', 'delivered', '2022-04-13 19:00:00', '2022-04-15 16:00:00', 1000, '89216457');

go

insert into Product values('hat',10,15.00015,'USA',2,8,'1234567890');

insert into Product values('AK-47',10,1500,'Vietnam',2,20,'1234567890');

insert into Product values('Cowboy T-Shirt',10,5,'Denmark',4,100,'1234567890');

go

insert into [Copy] values('2022-03-25 10:00:00', 100, 1);

insert into [Copy] values('2022-05-02 14:00:00', 500, 2);

go
```



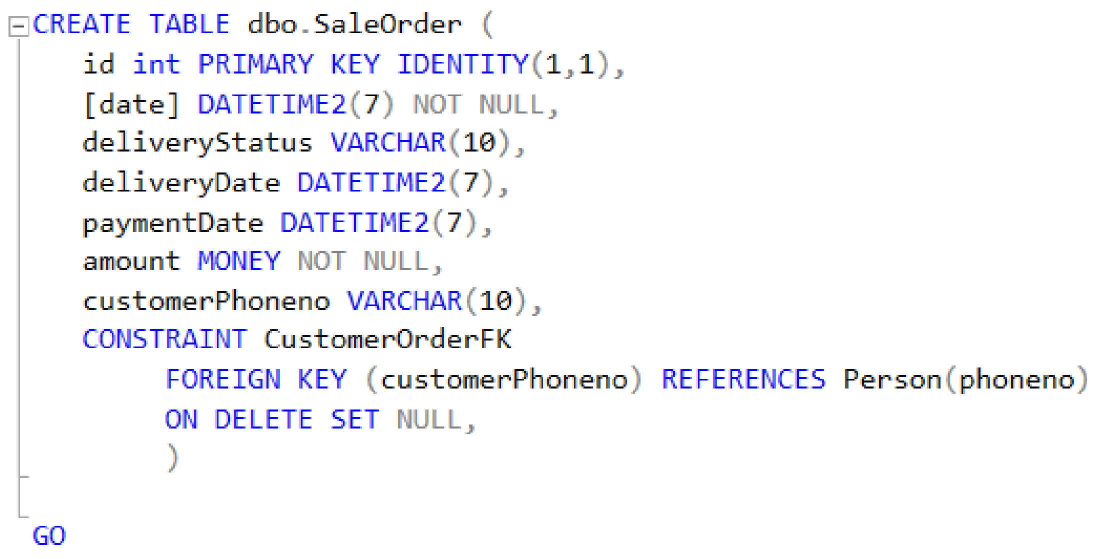
```sql
CREATE TABLE dbo.SaleOrderLine (
    quantity int NOT NULL,
    productId int,
    saleId int,
    PRIMARY KEY(productId, saleId),
    CONSTRAINT SaleOrderLineFK
        FOREIGN KEY (saleId) REFERENCES SaleOrder(id)
        ON DELETE CASCADE,
    CONSTRAINT CopySaleOrderLineFK
        FOREIGN KEY (productId) REFERENCES Product(id)
        ON DELETE CASCADE,
    )

GO

CREATE TABLE dbo.Clothing (
    size VARCHAR(5) NOT NULL,
    color VARCHAR(10) NOT NULL,
    productId int,
    CONSTRAINT ClothingProductFK
        FOREIGN KEY (productID) REFERENCES Product(id)
        ON DELETE SET NULL,
    )

GO

CREATE TABLE dbo.Equipment (
    [type] VARCHAR(20) NOT NULL,
    [description] VARCHAR(100) NOT NULL,
    productId int,
    CONSTRAINT EquipmentProductFK
        FOREIGN KEY (productID) REFERENCES Product(id)
        ON DELETE SET NULL,
    )

GO
```



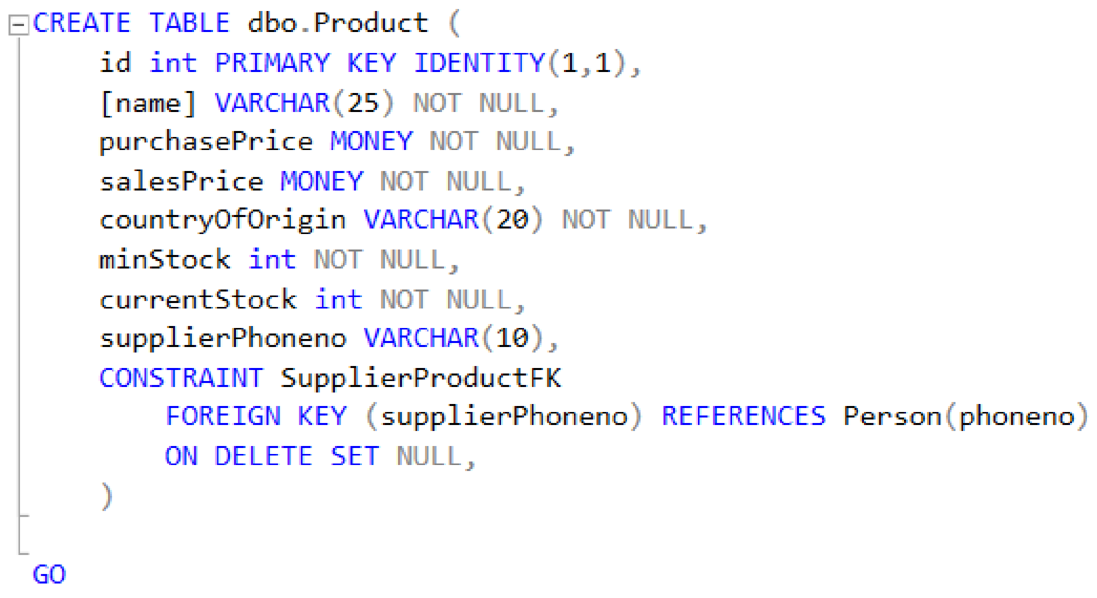
```sql
CREATE TABLE dbo.[Copy] (
    copyId int PRIMARY KEY IDENTITY(1,1),
    rentDate DATETIME2(7) NOT NULL,
    rentPrice MONEY NOT NULL,
    productId int,
    CONSTRAINT ProductCopyFK
        FOREIGN KEY (productId) REFERENCES Product(id)
        ON DELETE CASCADE,
    )

GO
```



```sql
CREATE TABLE dbo.RentOrderLine (
    quantity int NOT NULL,
    copyId int,
    saleId int,
    PRIMARY KEY (copyId, saleId),
    CONSTRAINT RentOrderLineFK
        FOREIGN KEY (saleId) REFERENCES SaleOrder(id)
        ON DELETE CASCADE,
    CONSTRAINT CopyRentOrderLineFK
        FOREIGN KEY (copyId) REFERENCES [Copy](copyId)
        ON DELETE CASCADE,
    )

GO
```

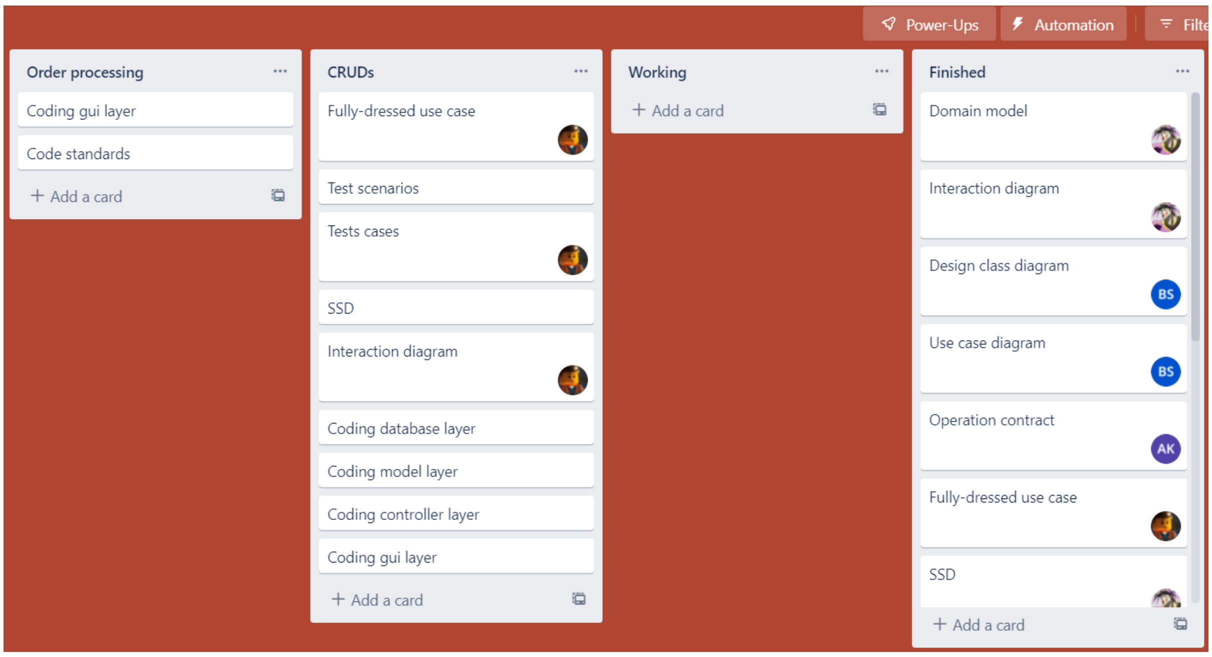


```sql
CREATE TABLE dbo.SaleOrder (
   id int PRIMARY KEY IDENTITY(1,1),
   [date] DATETIME2(7) NOT NULL,
   deliveryStatus VARCHAR(10),
   deliveryDate DATETIME2(7),
   paymentDate DATETIME2(7),
   amount MONEY NOT NULL,
   customerPhoneno VARCHAR(10),
   CONSTRAINT CustomerOrderFK
        FOREIGN KEY (customerPhoneno) REFERENCES Person(phoneno)
        ON DELETE SET NULL,
        )

GO
```



```sql
CREATE TABLE dbo.Product (
    id int PRIMARY KEY IDENTITY(1,1),
    [name] VARCHAR(25) NOT NULL,
    purchasePrice MONEY NOT NULL,
    salesPrice MONEY NOT NULL,
    countryOfOrigin VARCHAR(20) NOT NULL,
    minStock int NOT NULL,
    currentStock int NOT NULL,
    supplierPhoneno VARCHAR(10),
    CONSTRAINT SupplierProductFK
        FOREIGN KEY (supplierPhoneno) REFERENCES Person(phoneno)
        ON DELETE SET NULL,
    )

GO
```

Appendix2



Order processing	CRUDs	Working	Finished
Coding gui layer	Fully-dressed use case		Domain model
Code standards	Test scenarios		Interaction diagram
	Tests cases		Design class diagram (BS)
	SSD		Use case diagram (BS)
	Interaction diagram		Operation contract (AK)
	Coding database layer		Fully-dressed use case
	Coding model layer		SSD
	Coding controller layer		
	Coding gui layer		
+ Add a card	+ Add a card	+ Add a card	+ Add a card